Photogrammetric Computer Vision Lab1 Report

Professor Sohn

By SeyedMostafa Ahmadi

September
2021

my_imfilter Implementation

Pre-discussion

The first thing that came into my mind during the implementation of the cross correlation algorithm (my_imfilter) was that this would be computationally costly. So, What can I do? I started to develop two separate but similar algorithms. The first one is the one every beginner programmer would do, but the second one uses numpy broadcasting feature.

First Implementation

Parameters description

I have defined a function that takes 3 parameters: image, filter, and mode. Image and filter are the principal parameters, and mode is necessary for determining that the result image's resolution is the same or not. (same or valid)
Note that image and filter parameters should both be 2d matrices and for images with 3 colour channels we do this procedure 3 times. (one per each channel)

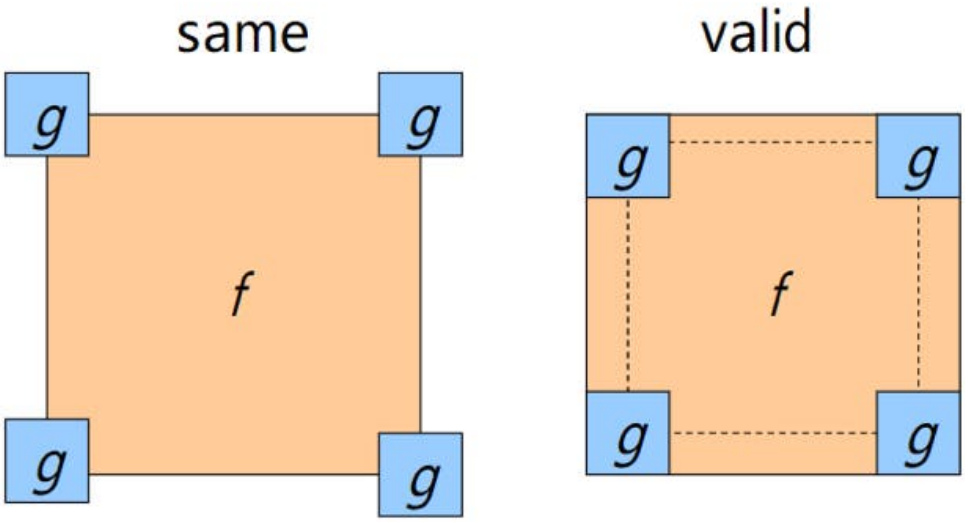

Figure 1. Two types of filtering that I implemented

---

procedure

If mode equals `same`, I add a zero padding to up/down and right/left of the original image based on the filter matrix shape. If mode equals `valid` we do not add any padding.
Then we make a loop on each index of the original matrix (not paddings!). each index will be the center of a sub-matrix that we name it A. Then we get the dot product of A and the filter (or kernel) matrix and put the result value in the same index of the Result matrix. Simple!

Second Implementation

Parameters description and the padding is just like above.

Figure 3. Original image of bottle

Figure 4. Bottle without background

Procedure

We make a 4d tensor out of 2d image in this way. We name the input a 2d image's matrix as A and the 4d tensor as B. We define B in this way:
B[i,j] = A[neighbourhoods of (i,j) element in shape of filter] for every (i,j) index in matrix A.
Then we multiply B to filter, and we name the result matrix C. (B is 4d and filter is 2d, but python broadcasts filter into a 4d tensor, and there will be no errors!)
Then we sum up the 3rd and 4th dimensions of C. The result is the same as the previous algorithm, but way faster!
Figure 2 shows the test results on my MacBook pro, but if you run algorithms_speed_test.py, it would be similar!

```
/Users/mostafa/PycharmProjects/computer_vision_course_lab1/ve
broadcast 0.7234430313110352 seconds
without broadcast 2.680145740509033 seconds

Process finished with exit code 0
```

Figure 2. Run time difference in two different cross correlation algorithms

This is a significant improvement if we use my_imfilter in a bigger scale of inputs!

Suggestion

For further usages, we can use jax (from google). Jax has its own numpy and its related operators but it can run on gpu. This way we can achieve very faster results. (but I do not have gpu on my MacBook pro)

Note on high and low frequency

Technically, high and low-frequency filtering is something that would be better to describe using Fourier Transform. But, it is not covered in this class and is off-topic (I have passed signal processing course during my bachelor's degree, so I have it from that course). But, to have an intuition, we can say that the higher frequencies appear in the parts of an image that change a lot and are not visible from a far distance. And lower frequencies appear in the parts of an image that change a little (or do not change at all) and are visible from a far distance.
I took a picture from a bottle of water (figure 1) (you do not have better views in quarantine!) and dropped out its background.

Then I generated following images. In the first image, only zero frequency signal appears.
In the second one, 0 and 1 frequencies appear.
In the third one: 0, 1, 2
…
And so on! (In the image below, I just selected some of them to show the progress faster!)
Look at these images that you can generate with running frequencies_test.py (images are for 0, 0 to 1, 0 to 9, 0 to 19, 0 to 99, and 0 to 999 respectively left to right and up to bottom):

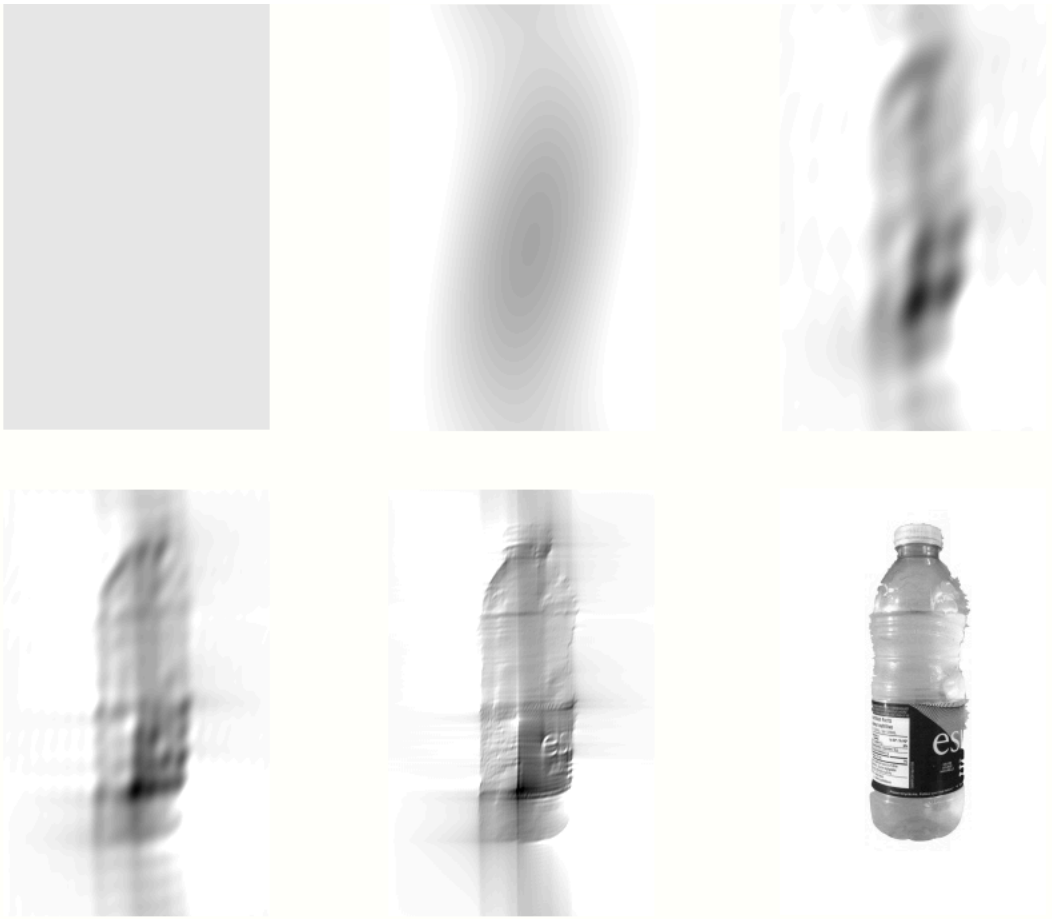

Figure 5. Appearance of different frequencies in bottle image

Complete images are inside the directory with path “results/frequency/”.

test_filtering results

To see the results of filtering you can take a look at “results/filtering” or you can simply run test_filtering.py to generate the plots. I have provided an all-in-one image for the results that you can see below.

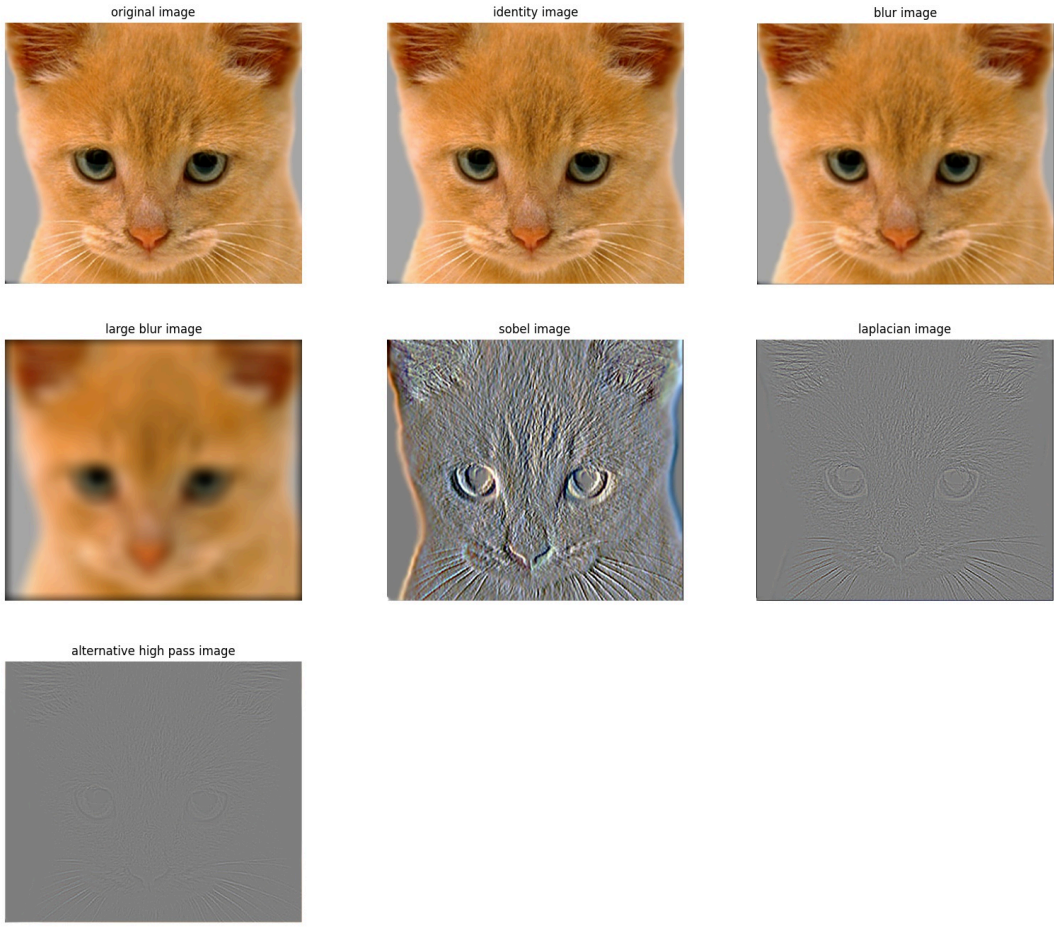

Figure 6. test_filtering results

hybrid_test Implementation

The implementation is quite easy as TAs described procedure. We need to combine high-frequency signals of one image with the low frequency of the other. The high frequency of an image can be achieved by subtracting its low frequency from it. Then we simply add to images. The input images were aligned and in the same resolution so I just read from the file and used it into the simple algorithm that used the low-frequency algorithm from the above section (I used the faster one).
You can run hybrid_test.py to see the results of the image that I aligned them. But, we should keep in mind that each pair of images had its own cutoff frequency that needed some experiments. I could use different cutoff frequencies for low frequency and high frequency as well, but I had a time shortage and could not test this one to check if can help the results significantly.

A tip I used for finding best cutoff value

After some minor amount of tests, it becomes trivial that if your cutoff value is higher than the best value, the image with high frequency becomes dominant and vice versa. (in the values lower than best value, image with low frequency is more dominant)

I simply used an algorithm in my experiments like “binary search”. Let me describe the procedure with an example. Assume we have a pair of images whose best cutoff frequency value is 9. To find this value, I always started with cutoff_frequency=20. Then, we will see that have passed high pass filter is more dominant. Then we choose its half; which is 10. We see that the result is better but still the image with high frequency is more dominant. We choose 5. In this turn image with a lower frequency is more visible and the higher one is barely visible. Then, we choose cutoff_frequency=(5+10)/2 which is not an integer but rounds to 7. Again, low is dominant. cutoff_frequency=(7+10)/2 which rounds to 8 and low is dominant. Finally cutoff_frequency=(8+10)/2 which is the best value for this pair.

To find the best value, you can use any algorithm you want! But, the one I suggested is the fastest way to find the best cutoff frequency which is found in O(log(n)) where n is 20. Actually, we should say that n is 32 to make the result logarithm an integer, which is 5! It means we can find the best value in 5 steps. But! In practice when we find a cutoff value very close to the best one, sometimes we, as human beings with lots of mistakes, can make mistakes and choose that value instead of the best.

vis_hybrid_image results

You can see the results of pair images below. Also I have added two more pairs! (you can also find these in “results/hybrid” or run vis_hybrid_image.py)

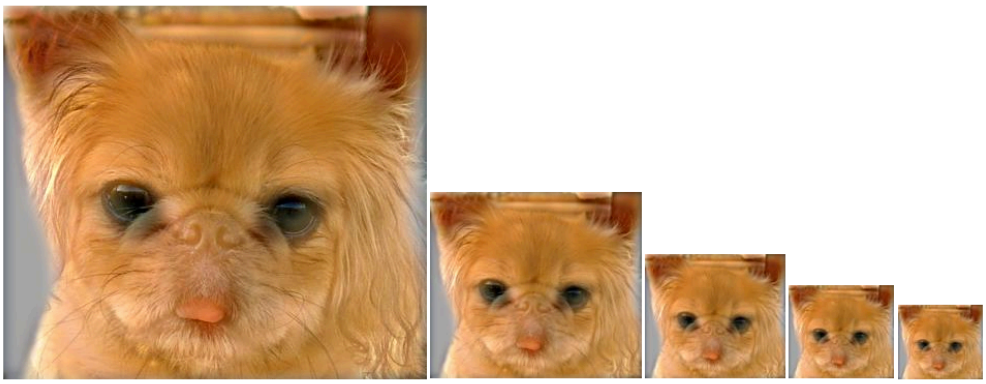

Figure 7. cat_dog pair, best cutoff = 7

Figure 8. einstein_marilyn pair, best cutoff = 4

Figure 9. bicycle_motorcycle pair, best cutoff = 9

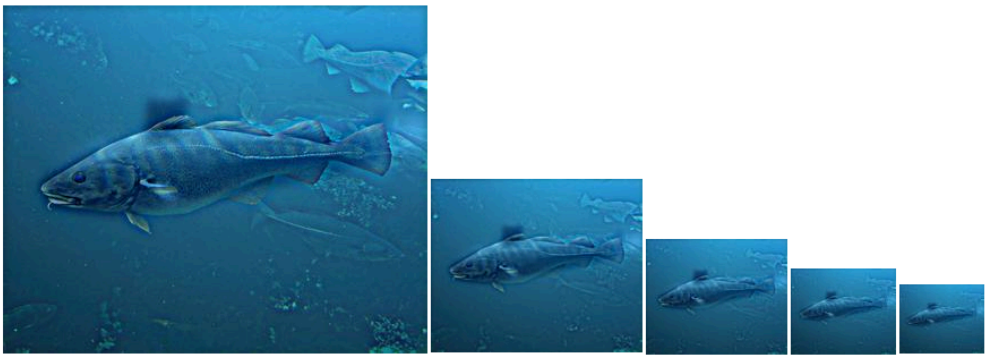

Figure 10. submarine_fish pair, best cutoff = 4

Figure 11. bird_plane pair, best cutoff = 6

I added these two. You can find the original images in the “figs” directory. The first image is the hybrid of the bottle we talked about and the starship of SpaceX. The second one is the hybrid of my childhood and teenage image.

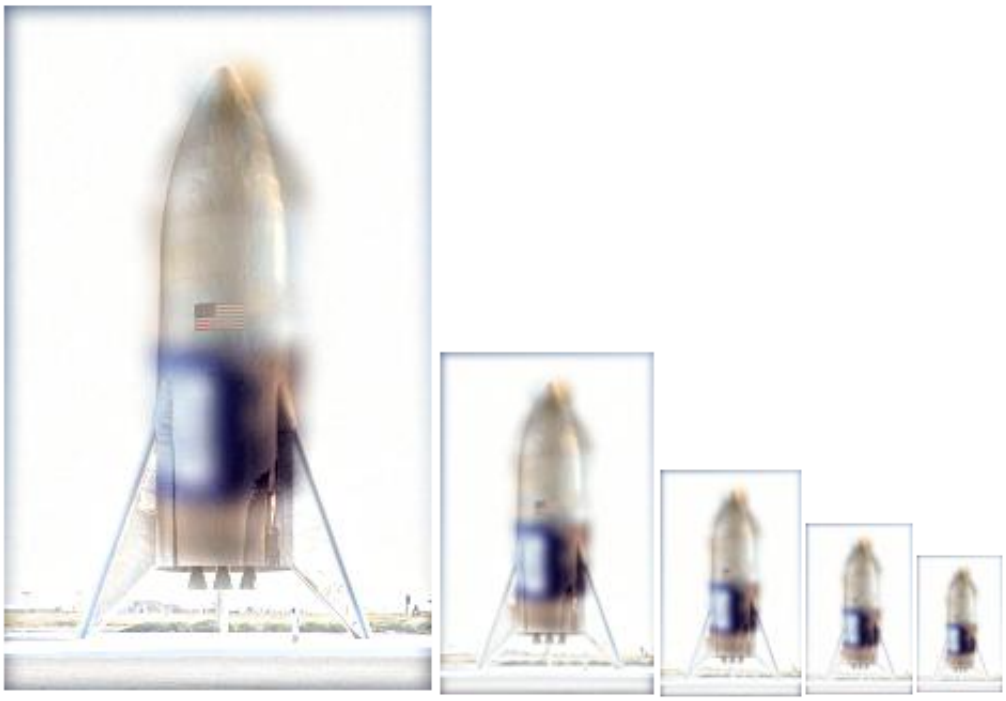

Figure 12. bottle_starship pair, best cutoff = 5

Figure 13. childhood_teenage pair, best cutoff = 3

That's it! I think I have done the project with all of its bonus points (and one more discussion about the low pass filtering algorithms!).

Thank you for your attention!